Cours de Systèmes d'Exploitation

Licence d'informatique

Synchronisation et Communication inter-processus

Hafid Bourzoufi

Introduction

Les processus concurrents s'exécutant dans le système d'exploitation peuvent être des processus coopératifs ou indépendants.

- Un processus est *indépendant* s'il n'affecte pas les autres processus ou ne peut pas être affecté par eux.
- Un processus qui ne partagent pas de données avec d'autres processus est indépendant
- Un processus est *coopératif* s'il peut affecter les autres processus en cours d'exécution ou être affecté par eux
- Un processus qui partage des données avec d'autres processus est un processus coopératif
- Les données partagées par les processus coopératifs peuvent se trouver en mémoire principale ou en mémoire secondaire dans un fichier
- Les accès concurrents à des données partagées peuvent produire des incohérences de données comme le montre l'exemple ci-dessous :

Exemple

Considérons la procédure suivante de mise à jour d'un compte bancaire d'un client. Pour simplifier, on considère qu'un compte est implémenté par un simple fichier qui contient le solde courant du client:

```
Procedure crediter_compte( entier numero_client, entier somme)
entier solde ;
debut
        solde=lire_dans_fichier(numero_client); /* lecture du solde dans le fichier du client */
        solde = solde + somme ;
        ecrire_dans_fichier(numero_client, solde) ; /* écrire dans le fichier le nouveau solde */
fin;
```

Supposons maintenant que cette même procédure est exécutée simultanément par deux processus P0 et P1 dans un système d'exploitation à temps partagé. Sans faire aucune hypothèse sur la durée du quantum, on peut faire les exécution suivantes :

P0 : crediter_compte(1233, 1000)	**P1 :** crediter_compte(1233, 500)
solde = lire_dans_fichier(1233) ;	
	solde = lire_dans_fichier(1233) ;
/* P0 bloqué car P1 s'exécute */	
	/* P1 bloqué car P0 s'exécute */
solde=solde+1000 ;	
Ecrire_dans_fichier(1233,solde) ;	
	solde=solde+500 ;
	Ecrire_dans_fichier(1233,solde) ;

Comme le montre cet exemple, le résultat final n'est pas le résultat escompté (*le client a perdu 500F dans cet affaire* ☹).

Sections critiques

Le problème dans le programme précédent est dû aux conflits d'accès au même fichier. Ces accès sont en lecture et en écriture. Evidemment, il ne peut y avoir de conflit si les accès aux données ne sont qu'en lecture seule.

- On appelle **section critique** la partie d'un programme où se produit le conflit d'accès

- Comment éviter ces conflits d'accès ?

  - Il faut trouver un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois
  - Il faut une **exclusion mutuelle** qui empêche les autres processus d'accéder à des données partagées si celles-ci sont en train d'être utilisées par un processus
  - Dans l'exemple précédent, il faut obliger le processus P1 à attendre la terminaison de P0 avant d'exécuter à son tour la même procédure

Pour une bonne coopération entre processus

Pour que les processus qui partagent des objets (données en mémoire centrale ou dans fichiers) puissent coopérer correctement et efficacement, quatre conditions sont nécessaires :

1. **Exclusion mutuelle** : deux processus ne peuvent être en même temps en section critique

2. **Famine** : aucun processus ne doit attendre trop longtemps avant d'entrer en section critique

3. **Interblocage** (*deadlock*) : aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres d'y entrer . La dernière section sera consacrée à l'étude de ce problème

4. **Aucun hypothèse** ne doit être faite sur les vitesses relatives des processus

Exclusion mutuelle par attente active

- Un processus désirant entrer dans une section critique doit être mis en atente si la section critique devient libre

- Un processus quittant la section critique doit le signaler aux autres processus
- Protocole d'accès à une section critique :

```
<entrer_Section_Critique>  /* attente si SC non libre */
<Section_Critique>         /* Un seule processus en SC */
<Quitter_Section_Critique>
```

- L'attente peut être :
  - **Active :** la procédure *entrer_Section_Critique* est une boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d'un processus en Section critique
  - **Non active :** le processus passe dans l'état endormi et ne sera réveillé que lorsqu'il sera autorisé à entrer en section critique

- Que contiennent les procédures *entrer_Section_Critique* et *quitter_Section_Critique ?*

1ère Solution : Masquage des interruptions

- Le moyen le plus simple est que chaque processus puisse masquer les interruptions avant d'entrer en section critique
  - ⇨ l'interruption horloge qui permet d'interrompre un processus lorsqu'il a épuisé son quantum (temps CPU), serait ignorée
  - ⇨ plus de commutation de processus

- Lorsqu'un processus quitte la section critique, doit restaurer les interruptions

- Solution dangereuse en mode utilisateur :

  - Si dans un processus, le programmeur a oublié de restaurer les interruptions, c'est la fin du système ☹

2ème solution : les variables de verrouillage

- Un verrou est une variable binaire partagée qui indique la présence d'un processus en Section Critique :
  - ⇨ si verrou = 0 alors la section critique est libre
  - ⇨ si verrou = 1 alors la section critique est occupée

- Procédure *entrer_Section_Critique () :*

```
void entrer_Section_Critique ()  {
if (verrou == 0) verrou=1 ;
else while (verrou == 1) ;    /* attente active */
verrou=1 ;
}
```

- Procédure *quitter_Section_Critique ()*

```
void quitter_Section_Critique () {
        verrou=0 ;
}
```

- L'exclusion mutuelle n'est assurée que si le test et le positionnement du verrou est ininterruptible (sinon le verrou constitue une section critique)

  - ⇨ Pour cela il faut disposer d'une instruction (test and set) qui teste et modifie le contenu d'un mot en mémoire centrale de façon indivisible.

3ème Solution : l'alternance

- On utilise une variable partagée (tour) qui mémorise le numéro du processus autorisé à entrer en section critique

- Exemple d'utilisation pour N processus :
  - ⇨

```
void entrer_Section_critique (int MonNumero)   {
while (tour !=  monNumero) ;  /* attente active */
}
```

  - ⇨

```
void quitter_Section_critique () {
tour=(monNumero +1) % N ;  /* au suivant ! */
}
```

- Avantage : simple et facile à utiliser
- Inconvénient : problème de **famine**, un processus possédant *tour*, peut ne pas être intéressé immédiatement par la section critique

Solution de *Peterson* (1981)

- Pour le cas de deux processus P0 et P1 :

```
#define FAUX      0
#define VRAI      1
#define N         2

int tour ;                  /* à qui le tour */
int interesse[N] ;          /* initialisé à FAUX */

void entrer_Section_Critique (int  process)  /* n° de processus : 0 ou 1*/
{
int autre ;
autre = 1-process ;
interesse[process]=VRAI ;       /* indiquer qu'on est intéressé  */
tour = process ;                /* lever le drapeau */
while (tour == process && interesse[autre] == VRAI) ;
}

void quitter_Section_Critique(int process)
{
        interesse[process]=FAUX ;
}
```

- Pourquoi l'exclusion mutuelle est assurée par cette solution?

Réponse : Considérons la situation où les deux processus appellent *entrer_Section_Critique* simultanément. Les deux processus sauvegarderont leur numéro dans la variable *tour*. La valeur sauvegardée en dernier efface la première. Le processus qui entrera en SC est celui qui a positionné la valeur *tour* en premier.

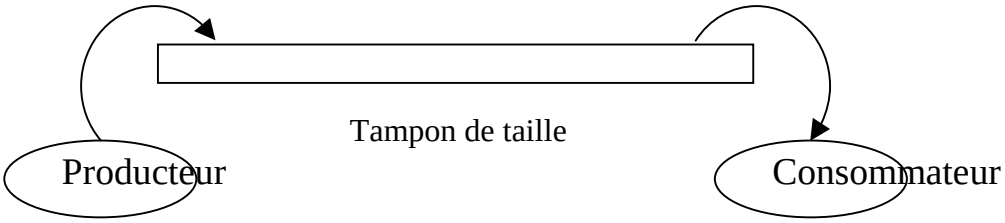
- Cette solution malgré qu'elle fonctionne bien, elle présente un gros inconvénient : elle basée sur l'attente active ; un processus ne pouvant entrer en SC utiliserait l'UC inutilement .

Solution évitant l'attente active

- **Idée :** un processus ne pouvant entrer en section critique, passe dans un état endormi, et sera réveillé lorsqu'il pourra y entrer.

  - ⇨ nécessite un mécanisme de réveil
  - ⇨ Le SE fournit deux appels système :
    - Sleep (dormir) qui suspend le processus appelant
    - Wakeup (réveiller) qui réveille le processus donné en argument

- Application au modèle Producteur/Consommateur



- ♦ Les deux processus coopèrent en partageant un même tampon
  - ⇨ Le producteur produit des objets qu'il dépose dans le tampon
  - ⇨ Le consommateur retire des objets du tampon pour les consommer

- ♦ **Conflits**
  - ⇨ Le producteur veut déposer un objet alors que le tampon est déjà plein
  - ⇨ Le consommateur veut retirer un objet du tampon alors que celui-ci est vide
  - ⇨ Le producteur et le consommateur ne doivent pas accéder simultanément au tampon

Code des processus producteur et consommateur

```
#define N 100          /* nbre d'emplacement ds tampon */

int compteur = 0 ;     /* nbre d'objets ds tampon */

void producteur () {
while (VRAI) {
        produire_objet(…) ;
        if (compteur == N) sleep () ;
        mettre_objet(…) ;
        compteur = compteur + 1 ;
        if (compteur == 1)
                wakeup(consommateur) ;
        }
}

void consommateur () {
while (TRUE) {
        if (compteur == 0) sleep() ;
        retirer_objet(…)
        compteur = compteur – 1 ;
        if (compteur == N-1)
                wakeup (producteur) ;
        consommer_objet(…) ;
        }
}
```

- Analyse de cette solution :
  1. L'accès à la variable *compteur* n'est pas protégé, ce qui peut entraîner des incohérences dans les valeurs prises par cette variable
  2. Réveils perdus : c'est le principal défaut de ce mécanisme. Un signal wakeup envoyé à un processus qui ne dort pas (encore) est perdu.